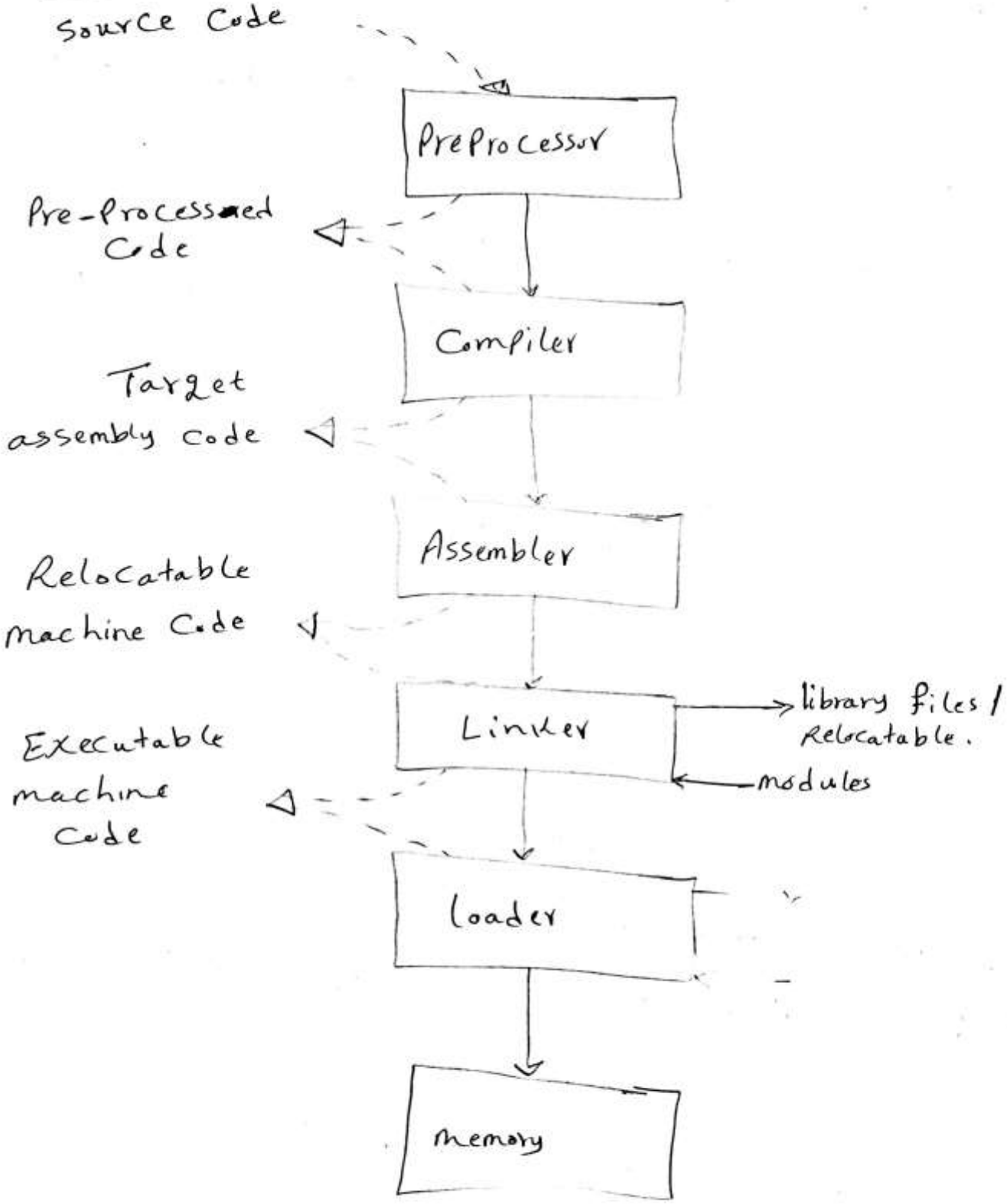
Lec 1

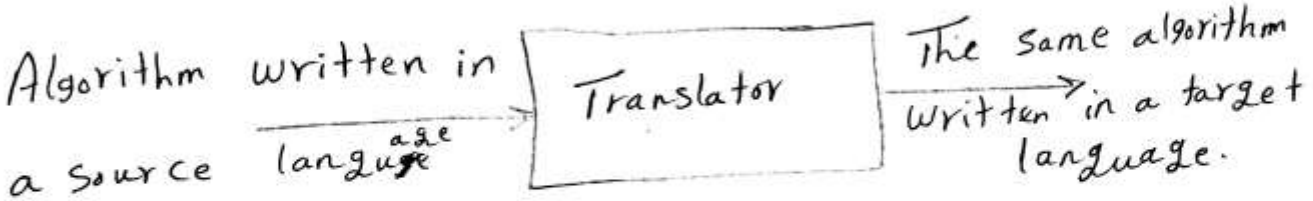
→ Language Processing System:-



→ Translators

* do the same functions of compilers.

* A program which accept a textual representation of an algrorithm expressed in a source language, and produce primary output a representation of the same Algorithm. expressed in another Languge.



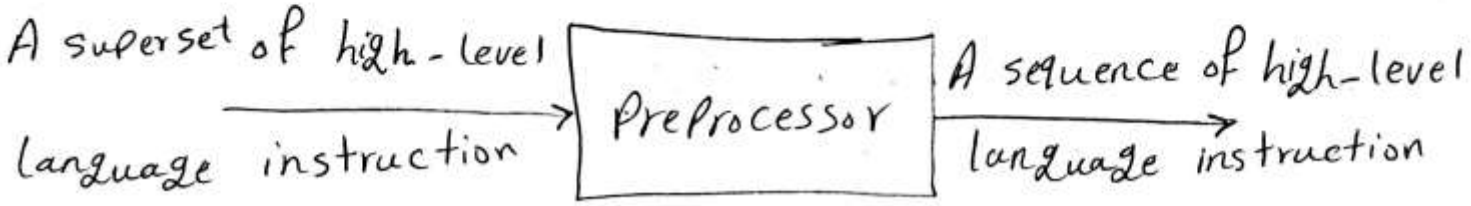
Types of translators

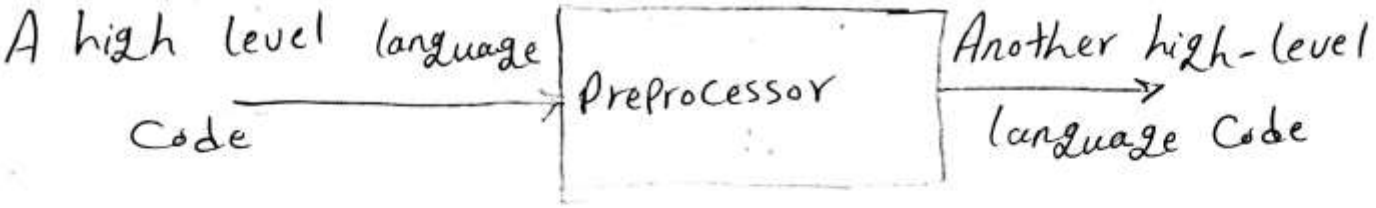
→ Pre Processor

* Considered as a part of compiler.
* Produces input for compilers.
* deals with macro-Processing, file inclusion, language extension, ... etc.

← يأخذ الـ (micro instructions) ويحوّلها ليفهمها.

* It can translate from high level language to another high level language.





Notes

→ Some compilers can give us executable code (ready) or assembly code (need some operations to be ready)

→ Translator has to get two programs do the same function to translate between them.

Types of translators "continued"

② Assembler :-

* translates assembly language Programs into machine code.

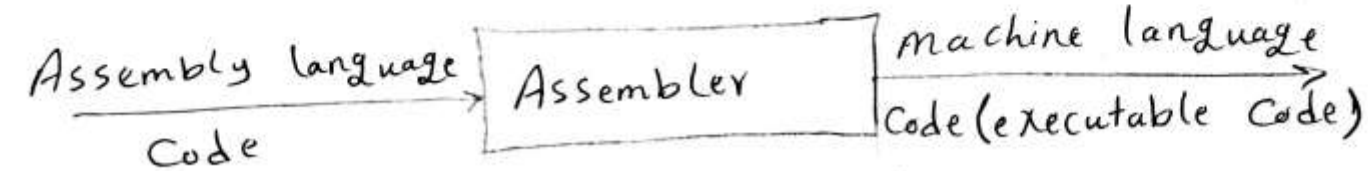
* It is (one-to-one)

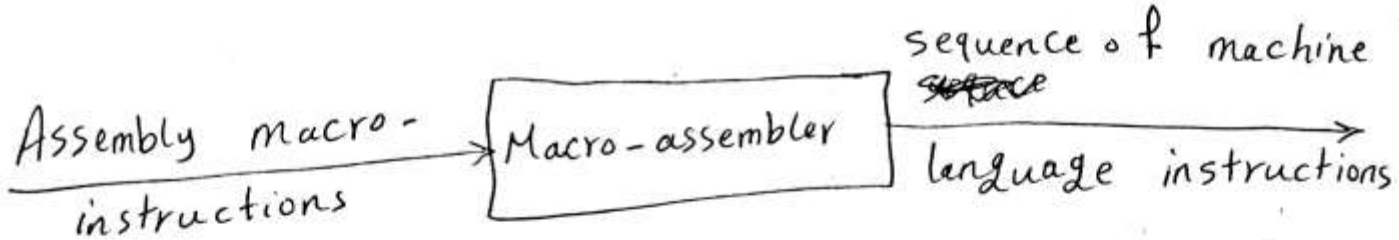
(one instruction) يعادل (assembly instruction) كل

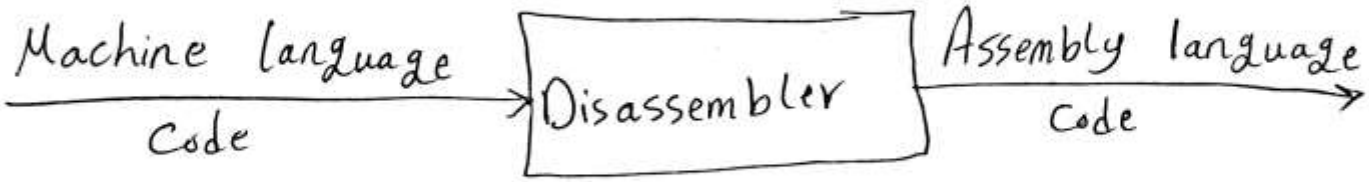
← مش وظيفته إنه يطلع (o/p)، ولا ينفذه

على (run) ← (machine) ويطلع o/p.

* It's o/p called object file, which Contains combination of machine instructions.



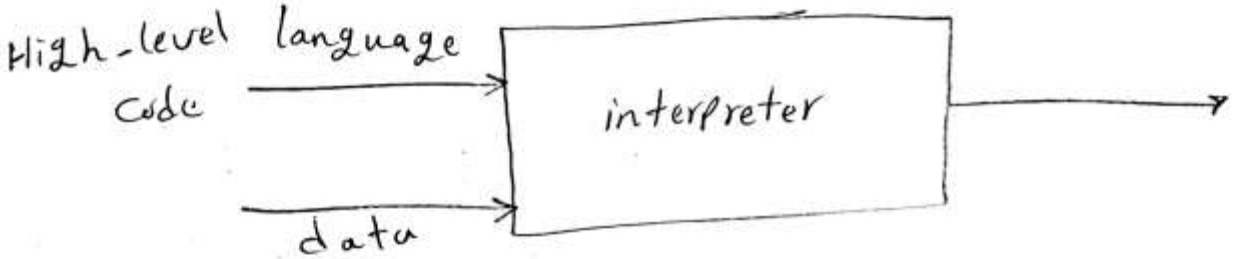




Types of translators "Continued"

3) interpreter :-

→ translates high-level language into low-level machine Language.



interpreter	compiler
→ translates program one statement at a time.	→ scan the entire program & translates it as a whole into machine code
→ No intermediate object code is generated hence are memory efficient.	→ Generates intermediate object code which requires linking hence requires more memory.
→ less amount of time to analyze source code → overall execution time is slower.	→ large amount time to analyze source code. → overall execution time is comparatively faster
→ Continue translating until program until first error is met, it stops. → Hence debugging is easy.	→ Generates error message only after scanning whole program. → debugging is comparatively hard.

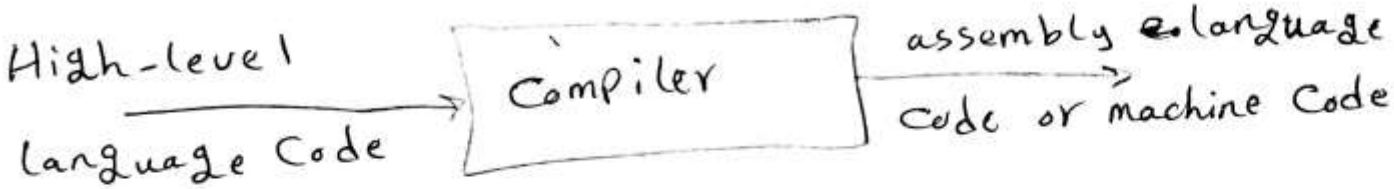
→ Programming language like Python use interpreter.

→ Programming language like C, C++ use Compilers

Compilers

→ it is a program Can read a program in one language - source language - and translate it into an equivalent program in another language - the target language.

→ report any errors in source program that it detects during the translation process.



→ classified with Considration of Construction

a) single pass b) Multiple pass

c) Load / Go.

→ classified with Considration of optimization:-

a) global b) Local.

* Cross Compiler

→ Compiler that runs on platform (A) and is capable of generating executable code for platform (B).

* Source-to-Source Compiler:-

→ A compiler takes the source code of one programming language and ~~translates~~ translates it into source code of another programming language.

* A Decompiler:-

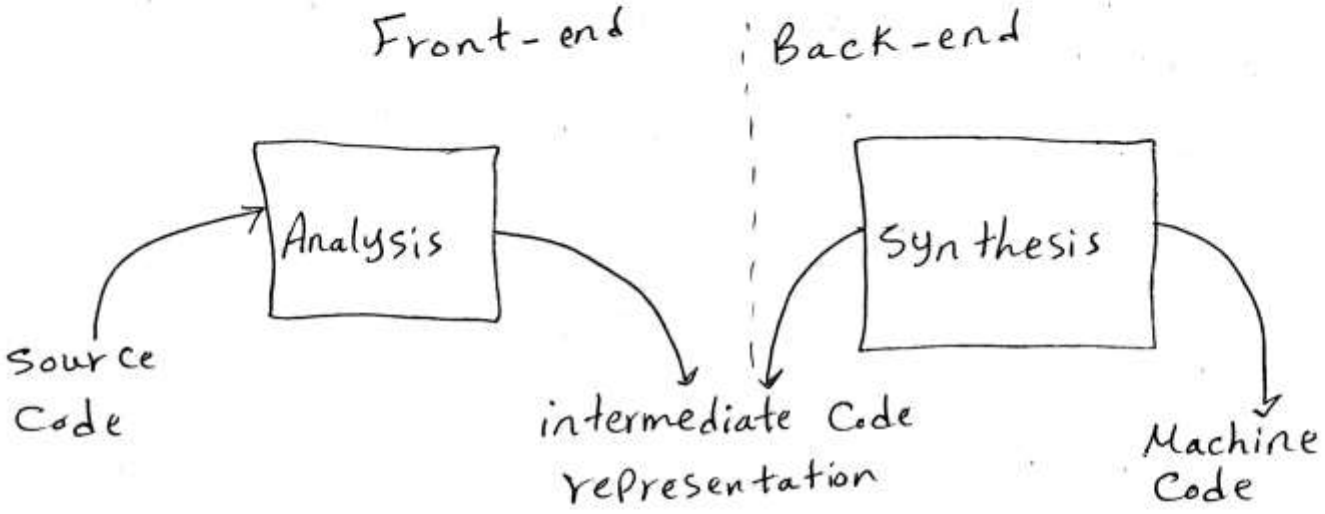
→ Program translates from low-level language (machine language) to a ~~A~~ higher-level language.

* $ Preprocessor is similar to source-to-source Compiler in the definition.

→ Compilers ~~is~~ clas are classified with Considration of function:-

a) debugging b) optimization.

Compiler Architecture



* Analysis Phase

→ Known as front-end of the Compiler.

→ it reads source program, then devides it into core parts, and then check for lexical, grammer and syntax errors.

→ Analysis go in three parts

a) lexical (scanner)

b) syntax.

c) semantic.

→ it generates an intermediate representation of the source program and symbol table, which should be fed to synthesis phase as input.

* synthesis Phase

→ Known as the back-end of the Compiler.

→ Generates the target program with the help of intermediate source code representation and symbol table.

front end → Analysis
→ intermediate code representation.

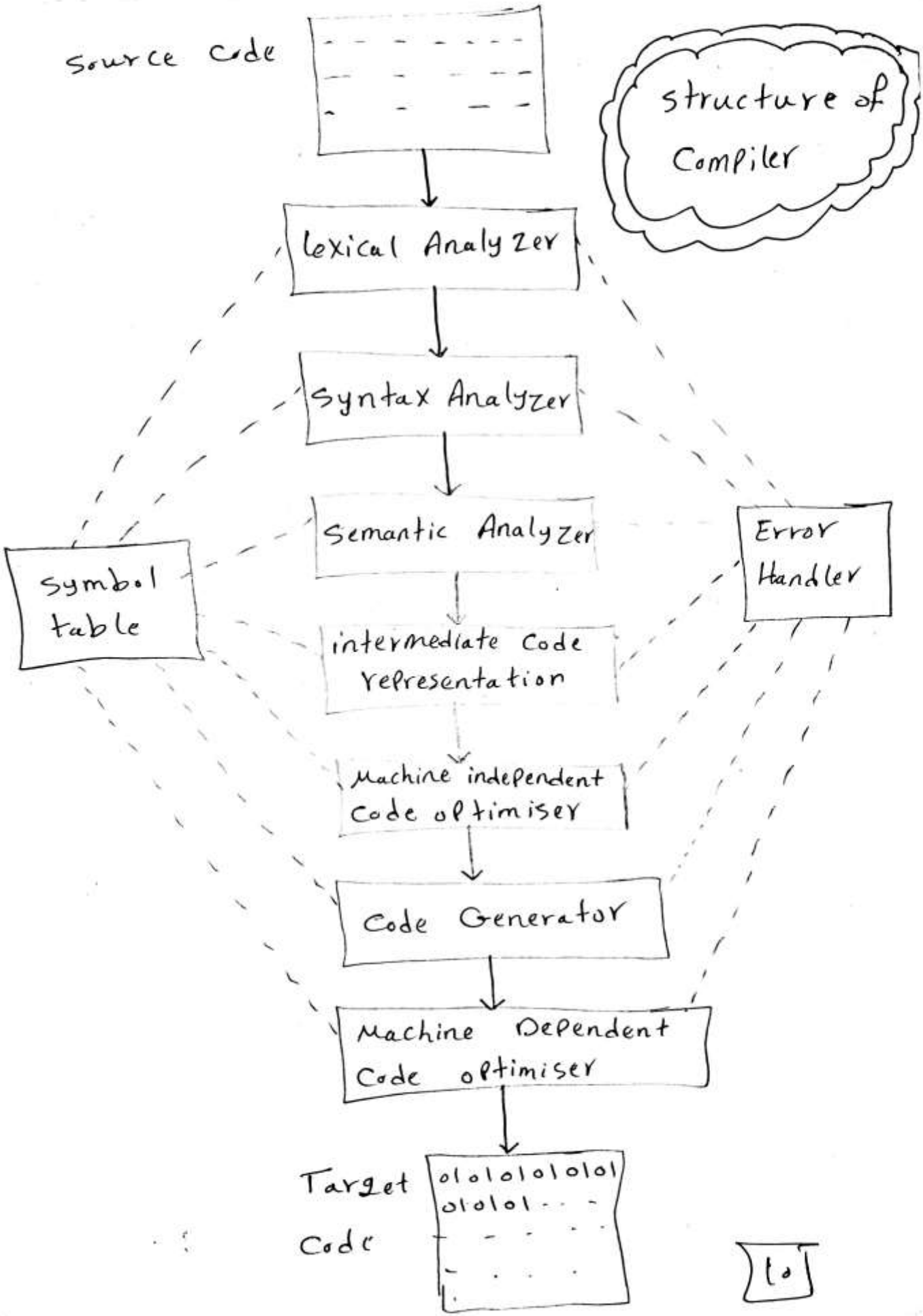
~~back end~~
(back end) , (front end) ← القياس ما بين
هو علاقته بال (machine).

back-end → if we work with local optimization.

~~syntax~~ machine independent → global optimiser
machine dependent → local " .

* phases we will study more

* lexical
* syntax
* Code Generator.
* machine dependent Code optimiser.


Source Code
structure of Compiler
lexical Analyzer
Syntax Analyzer
Semantic Analyzer
symbol table
Error Handler
intermediate Code representation
Machine independent Code optimiser
Code Generator
Machine Dependent Code optimiser
Target Code
0101010101
01010

* lexical analysis "scanner"

→ First Phase of Compiler.

→ works as a text scanner.

→ scans the source code as a stream of characters and Convertes it into meaningful lexemes.

→ represents these lexemes in the form of ~~tokens~~ tokens as:

< token-name, attribute-value >

→ lexems must be meaningful as a language.

← ترى ال (Command) فيا معناه لو ال (Command) مش بيشتيله (Preprocessor).

→ takes statement by statement.

← ال tokens هنا يعني (word) لكن في البرنامج فقط.

This is a sentence.

→ is not trivial. Consider:

ist his ase nte nce.

→ Lexical analyzer divides program text into "words" or "tokens".

if x == y then z=1 ; else z=2 ;

Syntax Analysis or Parsing

→ it takes the token produced by lexical analysis as input and generates a parse tree.

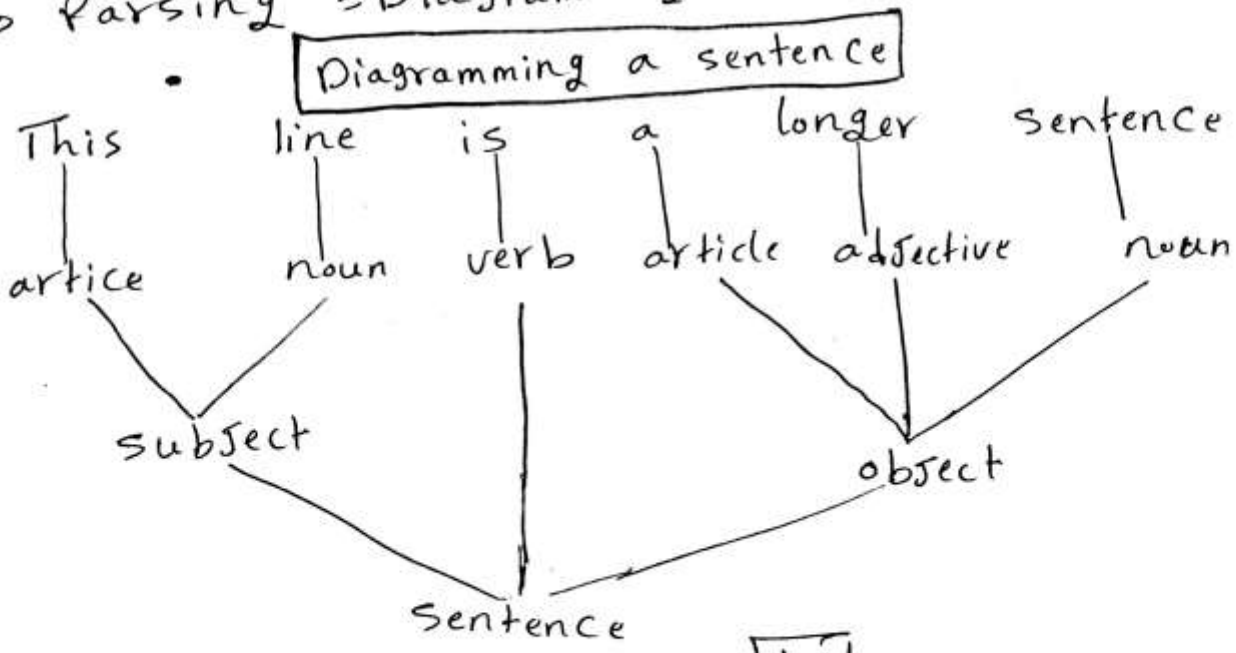
→ check source code for errors.

→ give us Parsal tree or atoms as o/p.

→ search for the sentence's structure.

→ once words are understood, next step is to understand sentence structure.

→ Parsing = Diagramming sentences

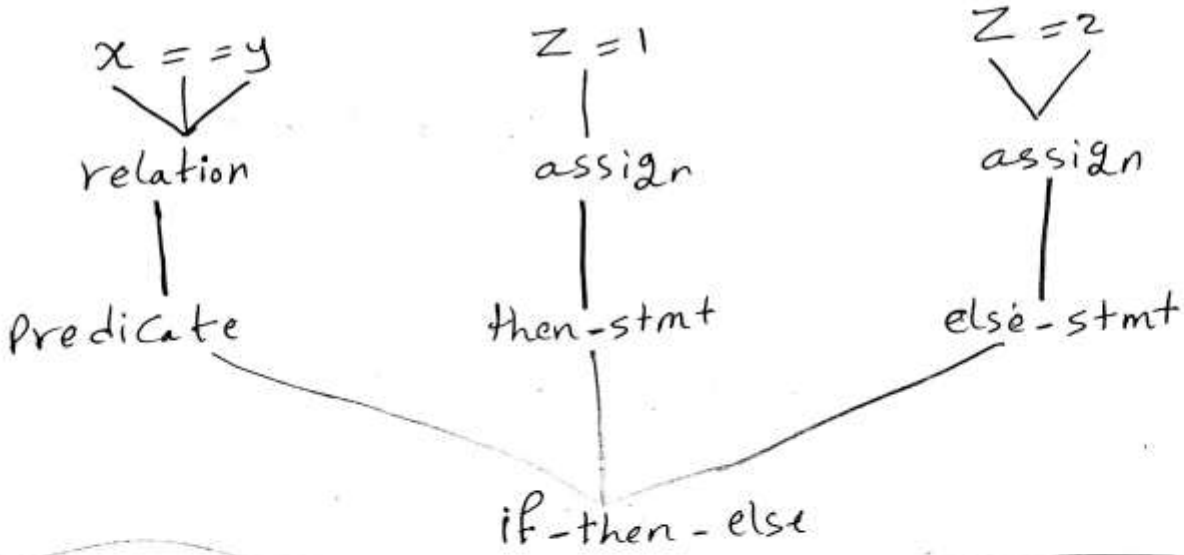


→ Parsing Program expressions is the same:-

Consider:

if x == y then z = 1; else z = 2;

Diagrammed



Semantic Analysis

→ once sentence structure is understood, we can try to understand "meaning".

→ compilers perform limited analysis to catch inconsistencies.

→ semantic checks wether the parse tree constructed follows the rules of language.

EX

Jack said Jerry left his assignment at home

→ what does "his" refer to? Jack or Jerry?

Even worse:

Jack said Jack left his assignment at home?

How many Jack are there?

which one left the assignment?

→ Semantic on Programming

```
{
  int Jack = 3;
{
  int Jack = 4;
  cout <<
  Jack;
}
}
```

→ This C++ Code prints "4"; inner definition is used.

→ Compilers perform many semantic checks besides variable bindings.

Ex: Jack left her homework at home

→ A "type mismatch" between her and Jack.

Optimization

→ Can be assumed as something that removes unnecessary code lines.

→ arranges the sequence of statements in order to speed up the program execution without wasting resources (cpu, memory)

→ Automatically modify programs so that they:

* Run faster.
* use less memory.
* in general, conserve some resource.

Ex

$X = Y * 0$ is the same as $X = 0$

Code Generation

→ usually produces assembly code.

→ takes the optimized representation of intermediate code and maps it to the target machine language.

→ translates the intermediate code into a sequence of re-locatable machine code.

→ ~~Sequence~~
→ A translation into another language
  * analogous to human translation.

Intermediate language

→ It represents a program for some abstract machine.
→ It is in between the high-level language and the machine language.
→ Intermediate code should be generated in such a way that it makes it easier to be translated into target machine language.

Note

Compiler Function

→ translate from high level language to low level language.
→ handling for error.
→ build symbol table.

Symbol table

→ data-structure maintained throughout all the phases of Compiler.

→ All the identifiers' names along with their types are stored here.

→ makes it easier for compilers to quickly search the identifer record and retrieve it.

→ used for scope management.

← بأخذ مكان في ال (memory)

← أي (variable) يخزن في (Symbol table)

يحجز مكان في الذاكرة ثم يتم تخزين ال(symbol table) في الذاكرة أي أنه يأخذ مكان في ال (memory).